References :

The following references are incorporated by references, into the present application:

1) C How to Program : Second Edition
H M DEITEL / P J DEITEL : Prentice Hall : ISBN 0-13-226119-7
2) C++ The Complete Reference : Second Edition
Herbert Schildt : McGraw Hill : ISBN 0-07-882123-1
3) Advanced Windows : Third Edition
Jeffrey Richter : Microsoft Press : ISBN 1-57231-548-2
4) Advanced topics in Data Flow Computing and MultiThreading :
Guang R Gao, Lubomir Bic, and Jean-Luc Gaudiot : IEEE Computer Society Press : ISBN 0-8186-6542-4

APPENDIX – A

SYNTAX AND SEMANTICS OF RCP STATEMENTS:

The syntax and semantics of the Rcp Statements, are presented below :

Syntactical Rules :

All Rcp statements begin with Exec Rcp token, and end with End_exec token. The Exec Rcp token is followed by the Rcp Statement name. Each Rcp statement contains a fixed number of parameters. Each parameter has a name, and the developer specifies a value for the parameter. Rcp Statements are not case sensitive, and upper and lower case characters are treated as same; however parameters are passed as is, and the interpretation is left to the host language.

The parms are specified as : PARM_NAME (Parm_value).

Statements can be specified on multiple lines, and can be broken whenever a keyword (token) ends;
For example, the parameter Num_of_elem can be specified on two lines as:

Num_of_elem
(10)

If the parameter value includes spaces or special characters it should be placed inside double quotes;
For example : Name (" This is a valid parameter value")

The actual pram value in this case is : This is a valid parameter value;

Please note the absence of quotes in the final value.

If the parameter value itself has double quotes then the escape sequence \" should be used to represent the double quotes.

For example : Name (" \" This is a quoted parameter value \" ")

The actual parameter value in this case is -> "This is a quoted parameter value";

Please note the presence of double quotes around the parameter value.

Classification of Rcp Statements :

Rcp statements are classified according to their usage :

1) Statements which can be specified in the .Resource definition file
2) Statements which can be specified in the program files
3) Statements which can be specified in both the .Resource definition file and the program files

It may be noted that executable Rcp Statements specified in the .Resource definition file will find support in Rcp_Init.cpp, which is automatically generated by the Rcp translator.

Semantic Rules :

Rcp statements can take variables or constants as parms. There are a few exceptions to this rule, and they are specified below :

1) Variable names and symbols defined to Rcp are indistinguishable. This problem is resolved by placing a '$' infront of the parameter value to inform that it is a host variable name.

For example, in the parameter Name (Q1), Q1 is a queue name which Rcp will try to translate, by looking into its symbol tables, whereas

In the parameter Name ($q2), q2 is regarded as a host language variable name and is passed to the host language compiler.

2) The symbol '#' is reserved for future use, and cannot be used as the first character of a variable name.

Rcp Error Check :

The Rcp Translator will generate code, which checks the return code of the Rcp library function corresponding to the Rcp Statement, and terminates further processing when the Rcp Statement fails. The developer can change this behavior and check return codes for specific failures by specifying the parameter Error_Check (YES) in the statement.

It may be noted that Abends cannot be bypassed, and the Rcp runtime will abend as soon as the node function returns control to the Rcp Runtime.

Rcp Statements :

1) Exec RCP [Resource definition file/Program file]
   Set Stmt_Num
   Psp (value)
   Cps (value)
   End_exec

This statement is used to change the statement numbers that are generated by

the program.

2) Exec RCP [Resource definition file/Program file]

Set Tab_Size

Count (value)

End_exec

This statement can be used to inform Rcp about the tab size of the source editor, so that the translated statements are indented correctly in the translated program.

3) Exec RCP [Resource definition file]

Create Frames

Count (value)

End_exec

This statement is used to create frames in the program.

4) Exec RCP [Resource definition file]

Create Threads

Min (value)

Max (value)

Frame (frame_num)

End_exec

This statement is used to create threads in the frame, note that Frame_num is optional, and if it is not specified the same number of threads are created in all the frames.

5) Exec RCP [Resource definition file]

Define Rcp_Gate

Name (Rcp_gate_name)

Qlist1 (queue_list_1)

Qlist2 (queue_list_2)

End_exec

This statement creates a Rcp_Gate and specifies the inputs/outputs of the Rcp_Gate. Either Qlist1 or Qlist2 parms has to be specified on this statement.

6) Exec RCP [Resource definition file]

Define Fctn

Name (fctn_name)

Qlist1 (queue_list_1)

Qlist2 (queue_list_2)

Rcp_Gate (rcp_gate_name)

End_exec

This statement creates a Rcp Node function and specifies the inputs/outputs of the function.

7) Exec RCP [Program file]

Run Pgm

Mode (security_mode)

End_exec

This statement starts the execution of the Rcp runtime library.The MODE parameter specifies the security mode in which the program will be executed.

8) Exec RCP [Program file]

Stop Run

End_exec

This statement will request Rcp to Stop the Frame. Frame will stop only after all functions have terminated.

9) Exec RCP [Program file]

Term Run

End_exec

This statement will terminate a Frame. All executing workers are terminated, immediately, and the frame is stopped immediately.

10) Exec RCP [Program file]

Term Pgm

End_exec

This statement will terminate all frames within the program.

11) Exec RCP [Resource definition file]

Define Queue

Name (queue_name)

Type (type)

Disp (queue_name)

End_exec

This statement will define a queue to the Rcp Implementation. It may be noted that only the name and type are specified at this time, the remaining details are captured from the Create statement.

12) Exec RCP [Resource definition file/Program file]

Create Queue

Name (queue_name)

Elem_size (value)

Num_of_elem (value)

Status (Ready / Not_Ready)

Error_Check (YES / NO)

Frame (Frame_num)

End_exec

This statement creates the queue, the queue can be readied upon creation by specifying Ready or Not_ready keyword in the status parm. Frame num will be ignored in Program file, since the system knows the frame in which the statement is running, via the Run_id; In Resource definition file the system lets the user specify the frame in which the queue has to be created; and FRAME_NUM_NULL implies all frames.

13) Exec RCP [Resource definition file/Program file]

Read Queue

Name (queue_name)

Elem (value)

Pointer (data_pointer)

Error_Check (YES / NO)

Frame (Frame_num)

End_exec

This statement is used to read the elements of a queue.

14) Exec RCP [Resource definition file/Program file]

Add Queue

Name (queue_name)

Elem_size (value)

Pointer (data_pointer)

Status (Ready / Not_Ready)

Error_Check (YES / NO)

Frame (Frame_num)

End_exec

This statement is used to add an element to queue. The return value, if greater than zero, indicates the element number added to the queue.

15) Exec RCP [Resource definition file/Program file]

Create Queue_Array

Name (queue_name)

Num_of_elem (value)

Queue_elem_size (value)

Num_of_queue_elem (value)

Error_Check (YES / NO)

Frame (Frame_num)

End_exec

This statement will create a Queue_Array.

16) Exec RCP [Program file]

Rebind Queues

Error_Check (YES / NO)

End_exec

This statement will rebind the inputs and outputs of the function.

17) Exec RCP [Program file]

Release Queues

Error_Check (YES / NO)

End_exec

This statement will release the input and output bindings of the function. Note that the function should be dependent upon a Rcp_Gate.

APPENDIX - B

HISTORY AND NOTATION :

Resource control programming (RCP) is the result of my research in parallel computing which started in 1990. The aim of this research is to create building blocks similar to the building blocks of digital electronics like AND GATE and OR GATE. The Rcp Gate is the result of this research, and is considered as a building block, in the development of multithreaded applications.

Consider a Rcp Gate G1, which is controlling a node function N1, which has say 2 invocations. Further assume that QA1 is the input queue array and QA2 and QA3 are the output queue arrays of the Rcp Gate. Assume that L1 is the local ring to which the Rcp gate is connected. A simple way of describing this information is as follows :

{QA1} → G1{N1[2], L1} → {QA2[8], QA3[8]}

This form is called short hand form, and although it is less illustrative than the block diagrams, it is useful for describing simple configurations.

It may be noted that curly braces hold the lists, and square braces hold the sizes, and parentheses are used to denote a particular instance, like the invocation of a node function, which is denoted by N1(1), or a particular queue in the queue array, which is denoted as QA1(1), or QA2(5). The queue array sizes are shown only once.

In the absence of input queue arrays, the Rcp Gate is represented as :

G1 {N1[2], L1} → {QA1[8], QA2[8]}

In the absence of output queue arrays, the Rcp Gate is represented as :

{QA1} → G1{N1[2], L1}

APPENDIX - C

BINDING AND RCP ARCHITECTURE :

Consider an Rcp Gate G1, with the following configuration :

{Q1, QA1[8], QA2[8]} → G1{N1[2], L1} → {QA3[8], QA4[8]}

It may be noted that during binding, the Rcp gate G1 selects a set of input queues like :

QA1(2), QA2(2) or
QA1(5), QA2(5)

Where the number within the parenthesis is called the input queue index, and is same for all input queue arrays. The same concept holds for output queue arrays, and the output queues are identified by output queue index like,

QA3(4), QA4(4) or
QA3(7), QA4(7)

Thus binding uses simple association to identify the queues within the queue arrays. However there are many special cases in binding, which are discussed below.

It may be noted that in the above example, Rcp Gate G1 is controlling two invocations of node function N1. Assuming that both invocations bound at the same time, we can write the bindings as :

{Q1, QA1(2), QA2(2)} → N1(1) → { QA3(4), QA4(4)} and
{Q1, QA1(5), QA2(5)} → N1(2) → { QA3(7), QA4(7)}

The important point in the above example is that queue Q1 is bound to both invocations of node function N1. This is a feature of Rcp architecture, where different invocations get different queues from queue arrays, whereas they share the queues. It may be noted that queues can be specified only on the input side of the Rcp gates, and on output side every invocation gets its own set of queues.

Now let us consider a slightly more complex situation, where we have two Rcp gates G1 and G2, and let us assume the following configurations for G1 and G2.

{Q1, QA1[8], QA2[8]} → G1{N1[2], L1} → {QA3[8], QA4[8]}
{Q1, QA1[8], QA2[8]} → G2{N2[2], L2} → {QA5[8], QA5[8]}

In this case the two invocations of N1 and the two invocations of N2 together share the Queue Q1, whereas one invocation of N1 and one invocation of N2 will share queues contained in queue arrays.

For example, QA1(5) is read by either N1(0) or N1(1) and
QA1(5) is also read by either N2(0) or N2(1)

It may be noted that if queue Q1 is of type "input-output", then it can be updated by all four invocations at the same time.

It is possible to have the following configuration :
{Q1, QA1[8], QA2[8]} → G1{N1[2], L1} → {QA3[8], QA4[8]}
{Q1, QA1[8], QA2[8]} → G2{N2[2], L1} → {QA3[8], QA4[8]}

It may be noted that G1 and G2 are now connected to the same local ring L1. The only restriction is that consumers should not use QA3 or QA4 with some other queue array say QA5, however they can use QA3, and QA4 together or independently.

APPENDIX - D

BINDING AND BIND SEQUENCE NUMBER :

Consider an Rcp Gate G1, with the following configuration :

G1{N1[2], L1} → {QA1[8]}

Since there are no inputs, the Rcp Gate G1 will select all available output queues in the output queue array QA1, and binds them to itself, which comprises of storing the bindings in the bind table of the Rcp gate. When an output queue is bound, the Rcp gate obtains a unique sequential number form the local ring to which it is connected, and assigns the unique sequential number to the binding by storing it in the bind table entry. This unique sequential number is called the BIND SEQUENCE NUMBER.

It may be noted that the bind sequence number is required to track the inputs which arrive on the input side of an Rcp Gate. Since there can be multiple invocations running concurrently for the top level Rcp Gates, the lower level Rcp Gates can see their inputs out of order, however the BIND SEQUENCE NUMBER helps the receiving Rcp Gate to reorder the inputs, so that the original order is not lost.

Another major purpose of the bind seq number is to provide a layer over the queues, so that any available queue set may be used, for a particular bind seq num. For example, bind seq num 33, need not use the 1st queue in the output queue arrays(assuming that the bind table size is 32). Bind seq num 33, may use any output queue set that is available at that time, which may be 10.

It may be noted that BIND SEQUENCE NUMBER may not be misinterpreted as an equivalent for the token concept of the data flow computing. Since the tokens of data flow computing architecture and the BIND SEQUENCE NUMBER of Rcp architecture are quite different and serve very different purposes. It is very important to observe that BIND SEQUENCE NUMBERS are not used for matching, and that there is no concept of matching in the Rcp architecture. In other words, data flow tokens are used for matching, whereas Rcp BIND SEQUENCE NUMBERS are used for sorting. The stage two of the BIND_VIRTUAL_QUEUES function 3207, in fact rearranges the inputs it receives, and this can be considered as a special kind of sorting.

With regard to the lack of matching in Rcp architecture, it may be noted that the Rcp architecture cannot handle certain configurations, which are described below.

Assume Rcp Gate G1, and Rcp Gate G2, and Rcp gate G3 are defined with the following configurations.

G1{N1[1], L1} → {QA1[8]}
G2{N2[1], L2} → {QA2[8]}
{QA1, QA2} → G3{N3[1], L3} → {QA3[8]}

The configuration for Rcp gate G3 is invalid because the Rcp gates G1 and G2 are independent of each other, and their outputs are not synchronized.

Associative matching is very well known in prior art, and hence is not discussed in this document. Implementers may optionally fuse other well known concepts with Rcp architecture.

The life cycle of the BIND SEQUENCE NUMBER is provided below for convenience.

1) Local Ring maintains the BIND SEQUNCE NUMBER, so that multiple Rcp gates writing output to the same queue array, still get sequential numbers, since Rcp gates which have common output queue arrays are connected to the same local ring.
2) Rcp Gate locks the local ring and acquires the BIND SEQUENCE NUMBER, when it allocates an output queue set identified by the output queue index to the incoming non null input queue set, identified by the input queue index. The incoming input queue set will have a BIND SEQUENCE NUMBER (qualified as INPUT), which may be different from what is allocated by this Rcp Gate (qualified as OUTPUT), since null inputs are bypassed by the Rcp Gate.
3) The Rcp gate stores the binding info, the input/output queue indexes, and the BIND SEQUENCE NUMBER in the bind table entry.
4) When a node function invocation does a successful Rebind, the BIND SEQUENCE NUMBER (OUTPUT) is copied to the node function invocation structure.
5) When this node function invocation sets an output queue to ready state, the BIND SEQUENCE NMBER is copied from the node function invocation structure to the Bind Sequence table entry of the Queue Array, corresponding to the queue number.
6) When the queue is consumed by all consumers, the RESET_QUEUE function 3210, sets the BIND SEQUENCE NUMBER in the Bind Sequence table entry of the Queue Array to NULL.
7) The BIND SEQUENCE NUMBER in the node function invocation structure is copied over by a new value during next iteration.
8) The BIND SEQUENCE NUMBER stored in the Bind table of the Rcp gate is reset to NULL by the Unbind_Virtual_Queues 3209 function.

APPENDIX – E

RCP GATE EFFICIENCY AND LOAD BALANCING :

The theory behind Rcp gate efficiency is illustrated below :

To begin with it is important to note that the stage four of the Bind_Virtual_Queues function 3207, deals with three different variations of bind sequence numbers namely the next output bind sequence num field 2815, in the Rcp Gate node, and the next output bind sequence num field 1903, in the local ring node, and the output bind seq num 2906 in the bind node structure of the bind table entry.

The number of outputs alloacted by the Rcp gate, is stored in the next output bind sequence num field 2815 of the Rcp gate node structure. It may be noted that this value for the rcp gate only, where as the next output bind sequence num field 1903, in the local ring node, may contain a higher value and pertains to the number of outputs allocated by all rcp gates connected to the local ring. The output bind seq num 2906 in the bind node structure of the bind table entry, is an instance of the next output bind sequence num field 1903, in the local ring node.

It may be noted that we are interested in the next output bind sequence num field 2815 of the Rcp gate node structure, and this number represents number of outputs allocated by the rcp gate, and may be slightly more than the number of outputs processed, but serves as a good approximation.

The number of times a worker is assigned to the node function invocations of the rcp gate is stored in the num of worker assignments field 2806 of the Rcp

gate node structure. Since we know the sizes of the queue arrays of the Rcp gate, we can deduce the following,

a) The number of queues processed per each worker assignment, by dividing the output bind sequence num field 2815 by the number of worker assignments.
b) Dividing the number obtained above by the capacity of the queue arrays and multiplying by 100 gives the percentage of queues processed per each worker assignment, and is termed as the rcp gate efficiency.

The theory behind load balancing is a series of simple ideas, as explained below.

a) Node functions can have arbitrary loads which are unknown, however a node function invocation of heavier load will take longer time than a node function invocation of lesser load.
b) In view of the above, it can be stated that the input queues of a node function invocation of heavier load build up, whereas the input queues of a node function invocation of lesser load are rapidly consumed, hence they evaporate.
c) In addition, it can also be stated that a node function invocation of heavier load, keeps running for longer times, compared to a node function invocation of lesser load, which completes processing quickly.
d) In view of the above it can be stated, that the Rcp gate efficiency of the rcp gate controlling the node function invocations of lesser load, will be poor, as compared to the rcp gate efficiency of the rcp gate controlling the node function invocations of heavier load.
e) In view of the above, it can be deduced, that when a node function invocation is running, and if the Rcp gate efficiency is very high, and if

the inputs and outputs available are very high, then that node function invocation is of heavier load.

f) Similarly it can be deduced, that when none of the node function invocations of the rcp gate are running, , and if the Rcp gate efficiency is very low, and if the inputs and outputs available are very low, then the node function invocations of that rcp gate are of lesser load.

The Rcp architecture uses the same principles, and considers 75% of Rcp gate efficiency and 75% of input and output queues available, and one or more node function invocations running, as a sign of heavier load. Similarly, 25% of Rcp gate efficiency and 25% of input and output queues available, and none of the node function invocations running, as a sign of lesser load.

When heavier load is detected more node function invocations are dispatched if possible, and when lesser load is detected, the dispatching of node function invocations is temporarily bypassed until the percentage of input and output queues available is greater than 25%.

Automatic load balancing my be achieved by blindly setting node max invocations configuration parameter to 2 or more, for every node function in the application. The Rcp runtime will compute rcp gate efficiencies and will switch workers automatically towards node function invocations of heavier load.

APPENDIX – F

An implementation of the sample application described in the operation section, is provided here to illustrate the Rcp architecture in greater detail.

The implementation is based on Windows 2000 operating system, and utilizes C++ as the host language.

The listings are provided on CD-Rom and microfiche.